MPI Parallel Programming
Part I

P.Y. Wang

Department of Computer Science 4A5

George Mason University

Fairfax VA 22030-4444 U.S.A.

References

These overheads are taken from three references:

- Bill Gropp's *Tutorial on MPI*
- OSC's MPI Primer/Developing with LAM
- Peter Pacheco's *Parallel Programming with MPI*

What is MPI?

- A message passing library specification
  - utilizes message-passing model
  - not a compiler specification
  - not a specific product
- Targeted to parallel computers, clusters, and heterogeneous networks
- Designed to encourage the development of parallel software libraries
- Designed to provide access to advanced parallel hardware for
  - end users
  - library writers
  - tool developers

Motivation

- Message passing is a mature programming paradigm:
  well understood, efficient match to hardware, suits many applications
- Vendor systems not portable
- Portable systems primarily research projects:
  incomplete, lack vendor support, not always most efficient
- Few systems offer the full range of desired features:
  modularity (for libraries), access to peak performance, portability, heterogeneity, subgroups, topologies, performance measurement tools

Who and When

- Broad spectrum of people participated in the MPI specification effort: vendors, library writers, applications specialists and consultants from industry, academia, national laboratories
- Effort began at Supercomputing '92

Features of MPI

- General
  - Communicators combine context and group for message security
  - Thread safety
- Point-to-Point communication
  - Structured buffers and derived datatypes, heterogeneity
  - Modes: normal (blocking/nonblocking), synchronous, ready (to allow access to fast protocols), buffered
- Collective
  - Both built-in and user-defined collective operations
  - Large number of data movement routines
  - Subgroups defined directly or by topology

Features of MPI, continued

- Application-oriented process topologies
  - built-in support for grids and graphs (using groups)
- Profiling
  - hooks allow users to intercept MPI calls to install their own tools
- Environmental:
  - inquiry, error control

Features Not in MPI

- Non-message-passing concepts:
  - process management
  - remote memory transfers
  - active messages
  - threads
  - virtual shared memory
- MPI does not address these issues, but has tried to remain compatible with them

Is MPI Large or Small?

- MPI is large (125 functions)
  - MPI's extensive functionality requires many functions
  - Number of functions not necessarily a measure of complexity
- MPI is small (6 function)
  - Many parallel programs can be written with just 6 basic functions
- MPI is just right
  - One can access flexibility when it is required
  - One need not master all parts of MPI to use it

Two commonly used MPI (Unix-based) implementations (U.S.):

- MPI LAM (Ohio Supercomputing $\Rightarrow$ Notre Dame)
- MPICH (Argonne National Laboratory)
- See P. Pacheco's web site (http://cs.usfca.edu/mpi/) for others

An Introduction to MPI and LAM

- LAM Architecture
  - LAM runs on each computer as a single daemon (server) uniquely structured as a nano-kernel and hand-threaded virtual processes
  - The nano-kernel component provides simple message-passing, rendezvous service to local processes
  - Some of the in-daemon processes form a network communication system, transferring message to/from other LAM daemons on other machines
  - Some of the in-daemon processes are servers for remote capabilities such as program execution and parallel file access
  - The layering is distinct: nano-kernel has no connection with network subsystem which has no connection with the servers

- LAM is transparent to users and system administrators
- LAM provides hands-on control of the multicomputer, especially from the debugging perspective

Getting Started

Here is an SPMD MPI program:

```
#include "mpi.h"
#include <stdio.h>

int main(argc, argv)
   int argc;
   char *argv[];
{
   MPI_Init (&argc, &argv);   /* Initialize MPI state */
   printf ("Hello world\n");
   MPI_Finalize()             /* Clean up MPI state */

   return 0;
}
```

What does it do???

Quick Start

To run an MPI (LAM) program, you will do something like this:

```
0: set up your LAMHOME and PATH variables
1: hcc -o foo foo.c -lmpi  % compile program
2: vi hostfile             & list compute resources schema
3: recon -v hostfile       % test schema
4: lamboot -v hostfile     % start LAM
5: tping n0....            % ping nodes
6: mpirun -v -c xxx foo    % xxx is the number of copies
7: mpitask or mpimesg      % to trace
8: lamclean -v             % to kill orphan processes
9: wipe -v hostfile        % terminate LAM
```

Broadening the View

```
#include "mpi.h"
#include <stdio.h>

int main(argc, argv)
   int argc;
   char *argv[];
{
   int rank, size;
   MPI_Init(&argc, &argv);   /* Initialize MPI state */
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);   /* who am I ? */
   MPI_Comm_size(MPI_COMM_WORLD, &size);   /* how many in all? */
   printf ("Hello world! I am %d of %d\n", rank, size);
   MPI_Finalize();            /* Clean up MPI state */

   return 0;
}
```


Communicators

- An MPI process is born into the world with 0 or more siblings
- The initial group is called the world group
- Each process has a unique number or rank between 0 and $N - 1$
- A process sends a message to the destination rank in the desired group
- The source rank may or may not be specified
- Messages are further filtered by a user-specified (arbitrary) tag which the receiver may ignore

- A context is attached by MPI to every message
- The four main synchronization variables in MPI are the source rank, destination rank, tag, and context
- A communicator is an opaque MPI data structure that contains information about one group and one context
- A communicator is an argument to all MPI communication routines
- After a process is created and initializes MPI, three pre-defined communicators (in ”mpi.h”) are available
  - MPI_COMM_WORLD: the world group
  - MPI_COMM_SELF: group with one member- me
  - MPI_COMM_PARENT
    an intercommunicator between two groups: my world group and my parent group (examined later in dynamic processes)

Blocking Point-to-Point Communication

- Blocking means the routine does not return until the associated buffer may be reused
- Point-to-point means the message is sent by one process and received by one process
- Four types of send modes are possible:
  - standard
  - buffered
  - synchronous
  - ready

Standard mode

- The send completes when the system can buffer the message (it is not obligated to do so) or when the message is received

Buffered mode

- The send completes when the message is buffered in the application supplied space, or when the message is received

Synchronous mode

- The send completes when the message is received

Ready mode

- The send must not be started unless a matching receive has been started. Then send completes immediately

Standard Mode Blocking Point-to-Point

```
MPI_Send( void *buf, int count, MPI_Datatype dtype, int dest
          int tag, MPI_Comm comm);

MPI_Recv( void *buf, int count, MPI_Datatype dtype, int source,
          int tag, MPI_Comm comm, MPI_Status *status);
```

A Simple Example

```
#include <stdio.h>
#include "mpi.h"

int main(argc, argv)
  int argc; char *argv[];
  {
   int rank;         /* Rank of process */
   int size;         /* Number of processes */
   MPI_Status status;  /* Return status for receive */
   double data[100];   /* Storage for the message */
   int i, count;       /* Some local counters */

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);

   count = 100;
```



```
   if (rank == 0) {

     MPI_Recv(data, 100, MPI_DOUBLE, 1, 10, MPI_COMM_WORLD, &status);
     printf("Data received from process 1:\n");
     for (i = 0; i < count; i++) printf("%f\n", data[i]);

   } else {

     if (rank == 1) {
        for (i = 0; i < count; i++) data[i] = (double) i;
        MPI_Send(data, count, MPI_DOUBLE, 0, 10, MPI_COMM_WORLD);
     }
   }

   MPI_Finalize();
}
```

Elementary MPI Data Types (in C)

MPI_CHAR	MPI_SHORT	MPI_INT
MPI_LONG	MPI_UNSIGNED_CHAR	MPI_UNSIGNED_SHORT
MPI_UNSIGNED	MPI_UNSIGNED_LONG	MPI_FLOAT
MPI_DOUBLE	MPI_LONG_DOUBLE	MPI_BYTE

NOTE: Beware machine dependencies!

MPI_INT could occupy 4 bytes on one machine and 8 bytes on another

A message count of 1 for both the sender and receiver would, in one direction, always be correct.

The message in the opposite direction could cause the communication to fail (not enough bytes)

Exercise

Write an MPI program that sends a message around a ring of processors. Process 0 sends to Process 1, process 1 sends to process 2, etc, with the last process sending the message to process 0.

Wildcards

- The source rank and the tag can be ignored in the receive by using MPI_ANY_SOURCE and MPI_ANY_TAG (wildcards)
- If wildcard(s) are used, then
  - status.MPI_SOURCE will contain the sender's rank
  - status.MPI_TAG will tag given by the sender
- To query the actual length of the message received:
  MPI_Get_count (MPI_Status *status, MPI_Datatype dtype, int *count)
- To synchronize a message without receiving it:
  MPI_Probe (in source, int tag, MPI_Comm comm, MPI_Status *status)

A More General Example

```
#include <stdio.h>
#include "mpi.h"

int main(argc, argv)
  int argc; char *argv[];
  {
   int i, count;     /* Some local counters */
   int rank;         /* Rank of process */
   int size;         /* Number of processes */
   int source;       /* Rank of sender */
   int dest;         /* Rank of receiver */
   int tag;          /* Tag for messages */
   double data[100];   /* Storage for the message */
   MPI_Status status;   /* Return status for receive */
```

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
printf ("Process %d of %d is alive!\n", rank, size);
```



```
dest = size - 1;
source = 0;

if (rank == source) {
  count = 10;
  for (i = 0; i < count; i++) data[i] = (double) i;
  tag = 2001;
  MPI_Send(data, count, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD);

} else {
```

```
    if (rank == dest) {
       MPI_Recv(data, 100, MPI_DOUBLE, source, MPI_ANY_TAG,
                MPI_COMM_WORLD, &status);
       MPI_Get_count (&status, MPI_DOUBLE, &count);
       printf("count received is %d\n", count);
       printf("status.TAG is %d\n", status.MPI_TAG);
       printf("Data received:\n");
        for (i = 0; i < count; i++) printf("%f\n", data[i]);
    }
  }

  MPI_Finalize();
}
```


The Output

```
Process 0 of 5 is alive!
Process 4 of 5 is alive!
count received is 10
status.TAG is 2001
Data received:
0.000000
1.000000
2.000000
3.000000
4.000000
5.000000
6.000000
7.000000
8.000000
9.000000
Process 2 of 5 is alive!
Process 1 of 5 is alive!
Process 3 of 5 is alive!
```

One-to-All-Broadcast

- Problem: Process 0 has data to be replicated to processes $1, 2, \ldots, n-1$.
- Solution: Use PRAM approach of recursive doubling
  - $0 \Rightarrow 1$
  - $0 \Rightarrow 2, 1 \Rightarrow 3$
  - $0 \Rightarrow 4, 1 \Rightarrow 5, 2 \Rightarrow 6, 3 \Rightarrow 7$
  - $0 \Rightarrow 8, 1 \Rightarrow 9, 2 \Rightarrow 10, 3 \Rightarrow 11,$
    $4 \Rightarrow 12, 5 \Rightarrow 13, 6 \Rightarrow 14, 7 \Rightarrow 15$

- There are $\lceil \lg n \rceil$ stages to this distribution
- For each stage, some processes are senders and others are receivers
  - For stage j, if $2^j \leq \text{my\_rank} < 2^{j+1}$, then I received from $\text{my\_rank} - 2^j$
  - For stage j, if $\text{my\_rank} < 2^j$, then I send to $\text{my\_rank} + 2^j$
- The pseudo-code for the one-to-all-broadcast would be

  for $stage \leftarrow 0$ **to** $\lg n$ **do**
  if I receive **then** Receive(data,source)
  else if I send **then** Send(data, dest)
  endfor

The "Send" Routine

- Suppose the "Send(data,dest)" routine involves sending three values a, b, and n.
- Then we might specify it as

```
void Send(float a, float b, int n, int dest) {
   MPI_Send(&a, 1, MPI_FLOAT, dest, 0, MPI_COMM_WORLD):
   MPI_Send(&b, 1, MPI_FLOAT, dest, 1, MPI_COMM_WORLD):
   MPI_Send(&n, 1, MPI_INT, dest, 2, MPI_COMM_WORLD):
}
```

Why do we use three different message tags?

The Corresponding "Receive" Routine

```
void Receive(float* a, float* b, int* n, int source) {

   MPI_Status status;

   MPI_Recv(a, 1, MPI_FLOAT, source, 0, MPI_COMM_WORLD, &status):
   MPI_Recv(b, 1, MPI_FLOAT, source, 1, MPI_COMM_WORLD, &status):
   MPI_Recv(n, 1, MPI_INT, source, 2, MPI_COMM_WORLD, &status):
}
```

See the Pacheco textbook for the complete code listing (pages 59^{+})

Collective Communications

- A communication pattern that involves all the processes is called a collective communication.
- Examples:
  - one-to-all: broadcast and reduce
  - one-to-all personalized: scatter and gather
  - allreduce (reduce and broadcast)
  - allgather (gather and scatter)

These could be implemented in MPI so as to take logarithmic time (see Pacheco book).

Broadcast/Reduce

- MPI_Bcast (void *buf, int count, MPI_Datatype dtype,
  int root, MPI_Comm comm);
- MPI_Reduce (void *sendbuf, void *recvbuf, int count,
  MPI_Datatype dtype, MPI_Op op,
  int root, MPI_Comm comm);

Reduction combiners:

MPI_MAX	MPI_MIN	MPI_SUM
MPI_PROD	MPI_LAND	MPI_BAND
MPI_LOR	MPI_BOR	MPI_LXOR
MPI_BXOR	MPI_MAXLOC	MPI_MINLOC

Example: Approximating π

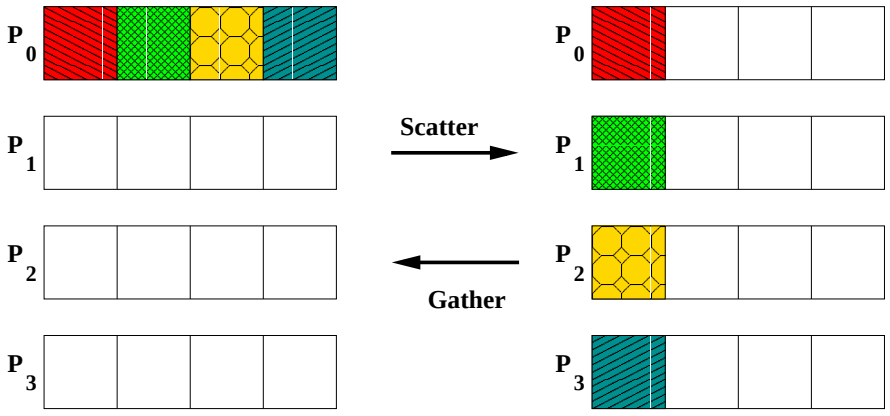
```
#include "mpi.h"
#include <math.h>

int main(argc,argv)
   int argc;
   char *argv[];
{
   /* This is a SPMD program to compute an approximation for pi */

   int done = 0, n, myid, numprocs, i, rc;
   double PI25DT = 3.14159265358979323846264 3;
   double mypi, pi, h, sum, x, a;

   /* Enter MPI and get world and my rank information */
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
```



```
   /* Continue to ask for number of intervals for the approximation */
   /* until a 0 is input by the user                                */
   while (!done)
   {
      /* The process with rank 0 asks for input value of n */
      if (myid == 0) {
         printf("Enter the number of intervals: (0 for quit) ");
         scanf("%d",&n);
      }

      /* Broadcast from process 0 to all processes */
      MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
      if (n == 0) break;

      /* Compute local approximation interval values using myid rank */
      h = 1.0 / (double) n;
      sum = 0.0;
      for (i = myid + 1; i <= n; i += numprocs) {
         x = h * ((double) i - 0.5);
         sum += 4.0 / (1.0 + x*x);
      }
      mypi = h*sum;
```

```
      /* All processes participate in a reduction to process 0 */
      MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

      /* Rank 0 process prints out the results */
      if (myid == 0)
         printf ("pi is approximately %.16f, Error is %.16f\n",
            pi, fabs(pi- PI25DT));
      }

      /* Exit MPI */
      MPI_Finalize();
}
```

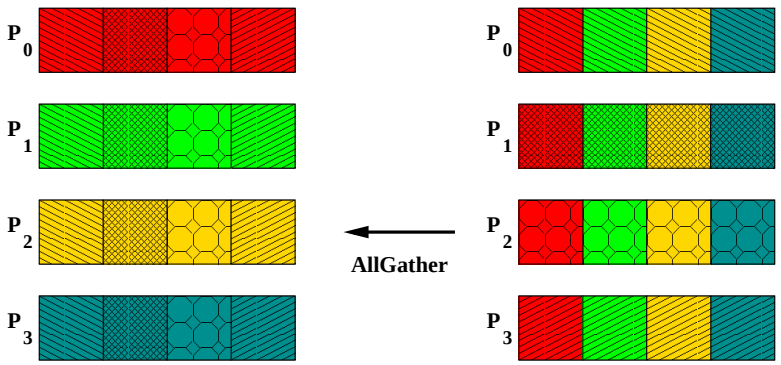

Scatter and Gather



- MPI_Scatter (void *sendbuf, int sendcount,
  MPI_Datatype sendtype, void *recvbuf,
  int recvcount, MPI_Datatype recvtype
  int root, MPI_Comm comm);
- MPI_Gather (void *sendbuf, int sendcount,
  MPI_Datatype sendtype, void *recvbuf,
  int recvcount, MPI_Datatype recvtype
  int root, MPI_Comm comm);

More Collective Communications

- MPI_Allreduce effects a reduce and then broadcast of the result
- MPI_Allgather:



- MPI_Allreduce (void *sendbuf, void *recvbuf,
  int count, MPI_Datatype dtype
  MPI_Op op, MPI_Comm comm);
- MPI_Allgather (void *sendbuf, int sendcount,
  MPI_Datatype sendtype, void *recvbuf,
  int recvcount, MPI_Datatype recvtype
  MPI_Comm comm);

These utilize the “hypercube” topology combining strategies

And More Collective Communications

MPI_Allgather	MPI_Allgatherv	MPI_Allreduce
MPI_Alltoall	MPI_Alltoallv	MPI_Bcast
MPI_Gather	MPI_Gatherv	MPI_Reduce
MPI_ReduceScatter	MPI_Scan	MPI_Scatter
MPI_Scatterv		

Versions ’v’ allow chunks of data to have different sizes

Buffering and Safety

- Note that tags are **not** used in collective communications
- It was implicitly assumed with MPI_Send that messages were *buffered*:
  i.e. that memory is set aside for storing messages before a "receive" has been executed
- Until process B calls MPI_Recv, the system doesn't know where B wants the message stored.
- If a system has no buffering, then A cannot send data until B is ready to receive
  - That is, memory is available for the incoming data;
  - This makes communication **synchronous**

- It is unsafe in MPI to assume that buffering occurs automatically; deadlock can occur
- Collective communications are non-blocking
- However, collective communication calls *behave* as synchronization points; otherwise we might have:

Time	Process A	Process B	Process C
1	MPI_Bcast x	local work	local work
2	MPI_Bcast y	local work	local work
3	local work	MPI_Bcast (y)	MPI_Bcast (x)
4	local work	MPI_Bcast (x)	MPI_Bcast (y)

Collective Barriers

- To synchronize the processes
  int MPI_Barrier (MPI_Comm comm);
- Should be used only if needed- can slow a program down
- Useful for timing programs in single-user scenarios
  double MPI_Wtime (void);

Timing a Program

```
double start, finish;

MPI_Barrier(comm)
start = MPI_Wtime();
  .
  . code being timed
  .
MPI_Barrier(comm)
finish = MPI_Wtime();

if (my_rank == 0)
   printf("Elapsed time = %e seconds\n", finish-start);
```